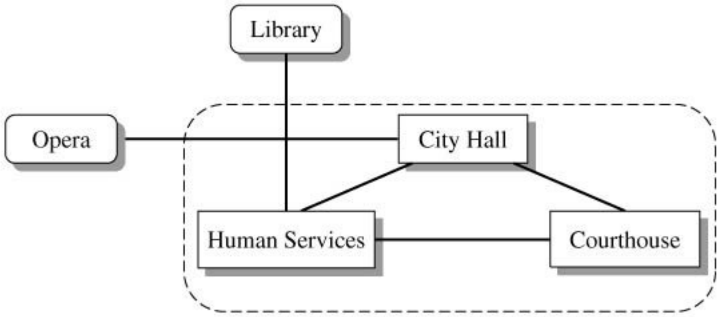
Graph

Terminologi Graph

- A graph consists of a set of *vertices* V, along with a set of *edges* E that connect pairs of vertices.
  - An edge e = (vi,vj) connects vertices vi and vj.
  - A *self-loop* is an edge that connects a vertex to itself. We assume that none of our graphs have self-loops.

Vertices = $\{v_1, v_2, v_3, …, v_m\}$
Edges = $\{e_1, e_2, e_3, …, {}_{en}\}$

Graph Terminology (continued)



Civic center graph.

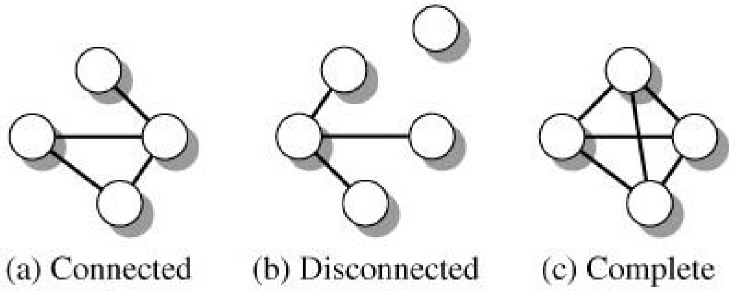
Graph Terminology (continued)

- The *degree* of a vertex is the number of edges originating at the vertex.
- Two vertices in a graph are *adjacent* (*neighbors*) if there is an edge connecting the vertices.
- A path between vertices v and w is a series of edges leading from v to w. The path length is the number of edges in the path.

Graph Terminology (continued)

- A *path* is simple if all its edges are distinct. A cycle is a simple path that starts and ends on the same vertex.
- A graph is *connected* if there is a path between any pair of distinct vertices.
- A *complete graph* is a connected graph in which each pair of vertices is linked by an edge.

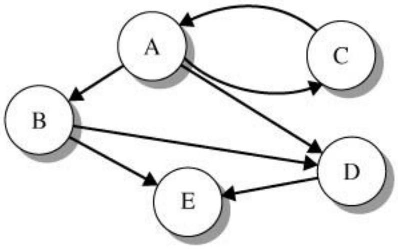
Graph Terminology (continued)



Graph Terminology (continued)

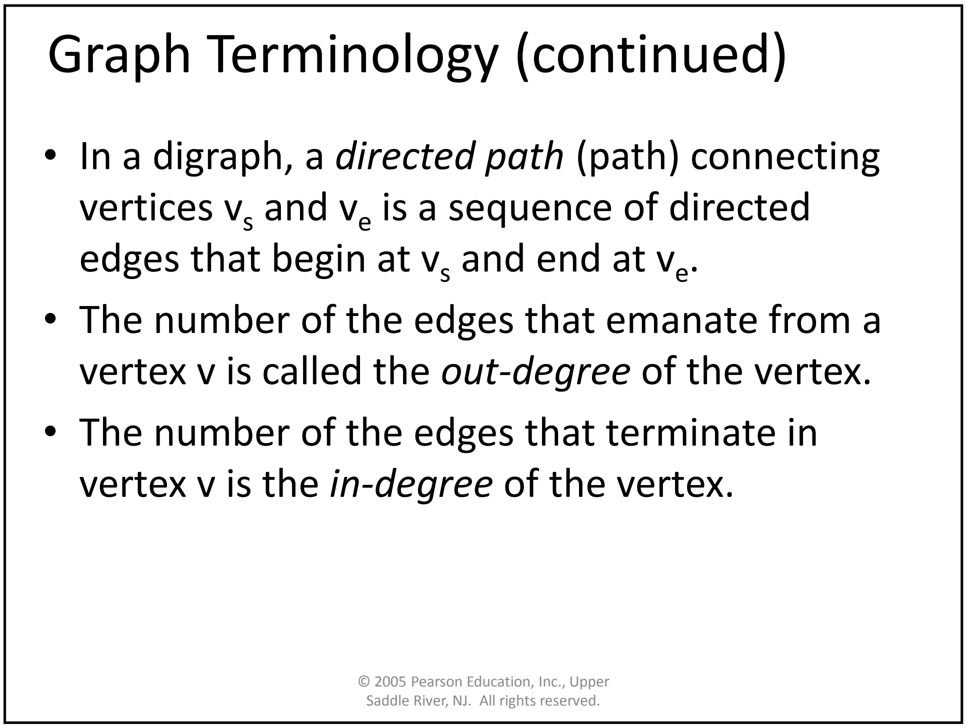
- A graph described until now is termed an *undirected graph*. Movement between vertices can occur in either direction.
- In a *digraph*, edges have a direction. There might be an edge from v to w but no edge from w to v.

Graph Terminology (continued)



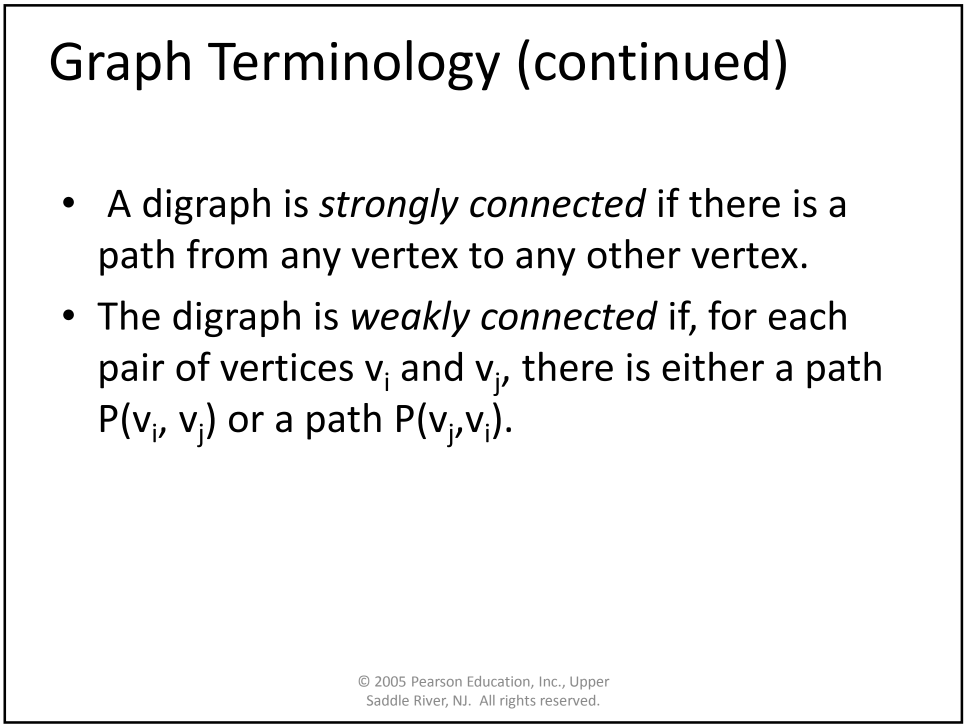
Vertices V = {A, B, C, D, E}
Edges E = {(A, B), (A, C), (A, D), (B, D), (B, E), (C, A), (D, E)}

Sample digraph with five vertices and seven edges.



Graph Terminology (continued)

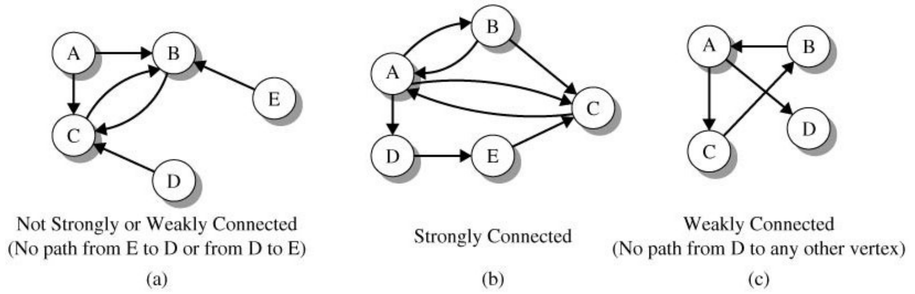
- In a digraph, a *directed path* (path) connecting vertices v_s and v_e is a sequence of directed edges that begin at v_s and end at v_e.
- The number of the edges that emanate from a vertex v is called the *out-degree* of the vertex.
- The number of the edges that terminate in vertex v is the *in-degree* of the vertex.



Graph Terminology (continued)

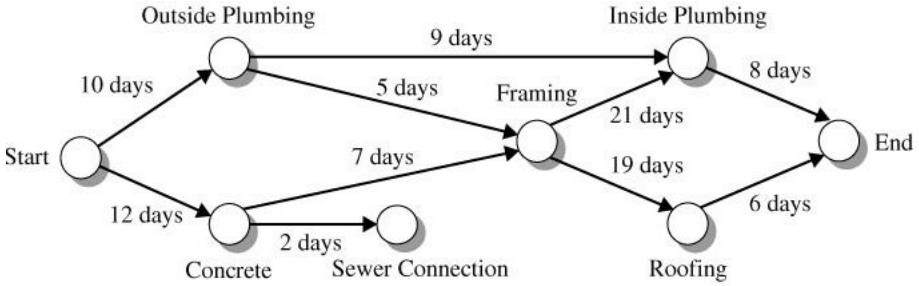
- A digraph is *strongly connected* if there is a path from any vertex to any other vertex.
- The digraph is *weakly connected* if, for each pair of vertices v_i and v_j, there is either a path $P(v_i, v_j)$ or a path $P(v_j,v_i)$.

Graph Terminology (continued)


A
B
C
D
E
Not Strongly or Weakly Connected
(No path from E to D or from D to E)
(a)
Strongly Connected
(b)
Weakly Connected
(No path from D to any other vertex)
(c)

Graph Terminology (concluded)

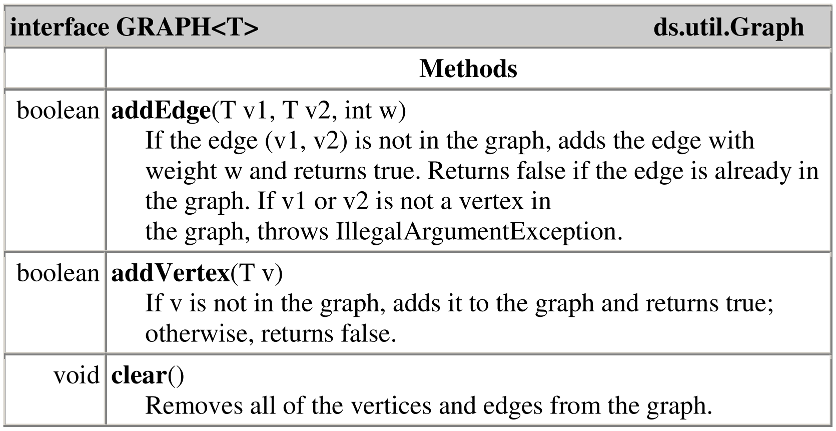
- An *acyclic* graph has no cycles.
- Each edge in a *weighted digraph*, has a cost associated with traversing the edge.



Creating and Using Graphs

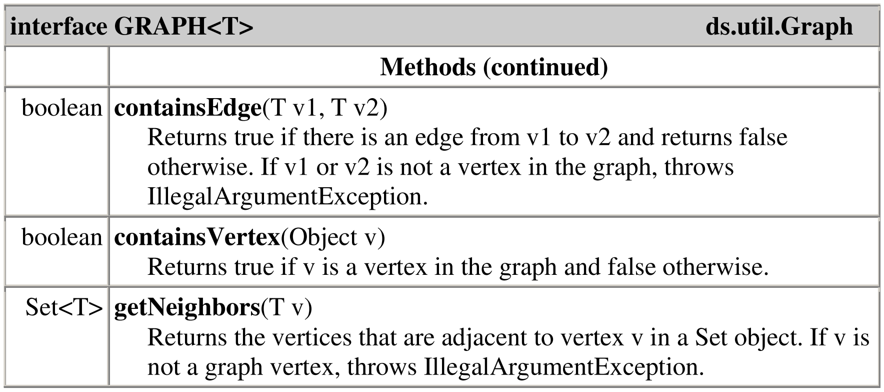
- The Graph interface specifies all basic graph operations including inserting and erasing vertices and edges.

Creating and Using Graphs (continued)



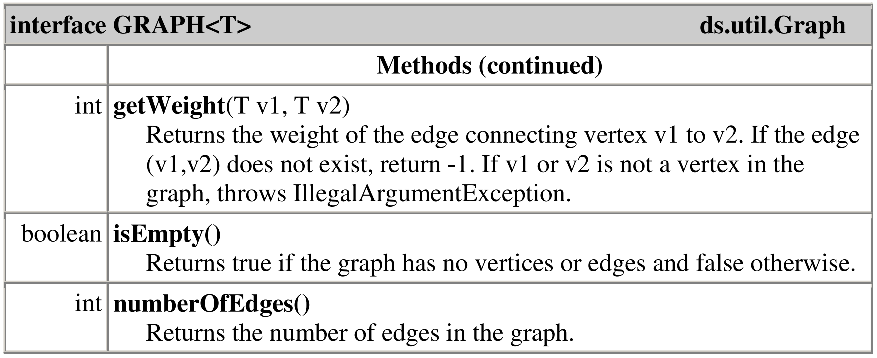
interface GRAPH<T>	**ds.util.Graph**
	Methods
boolean	**addEdge**(T v1, T v2, int w) If the edge (v1, v2) is not in the graph, adds the edge with weight w and returns true. Returns false if the edge is already in the graph. If v1 or v2 is not a vertex in the graph, throws IllegalArgumentException.
boolean	**addVertex**(T v) If v is not in the graph, adds it to the graph and returns true; otherwise, returns false.
void	**clear**() Removes all of the vertices and edges from the graph.

Creating and Using Graphs (continued)



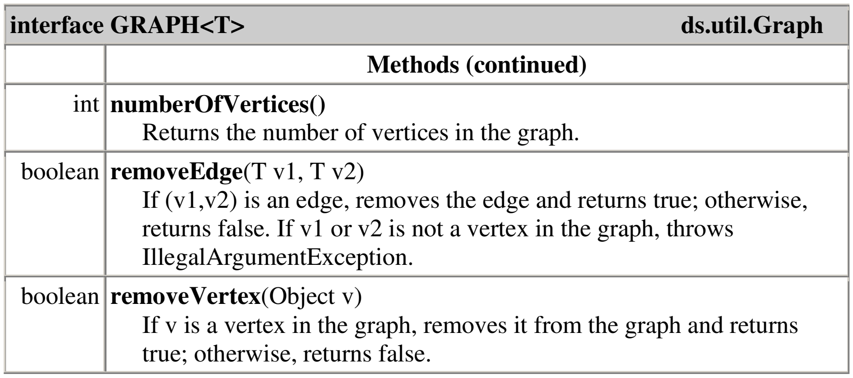
interface GRAPH<T>	**ds.util.Graph**
	Methods (continued)
boolean	**containsEdge**(T v1, T v2) Returns true if there is an edge from v1 to v2 and returns false otherwise. If v1 or v2 is not a vertex in the graph, throws IllegalArgumentException.
boolean	**containsVertex**(Object v) Returns true if v is a vertex in the graph and false otherwise.
Set<T>	**getNeighbors**(T v) Returns the vertices that are adjacent to vertex v in a Set object. If v is not a graph vertex, throws IllegalArgumentException.

Creating and Using Graphs (continued)



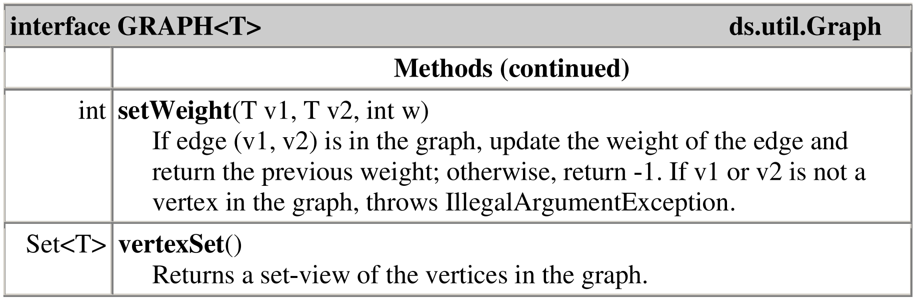
interface GRAPH<T>	**ds.util.Graph**
	Methods (continued)
int	**getWeight**(T v1, T v2) Returns the weight of the edge connecting vertex v1 to v2. If the edge (v1,v2) does not exist, return -1. If v1 or v2 is not a vertex in the graph, throws IllegalArgumentException.
boolean	**isEmpty**() Returns true if the graph has no vertices or edges and false otherwise.
int	**numberOfEdges**() Returns the number of edges in the graph.

Creating and Using Graphs (continued)



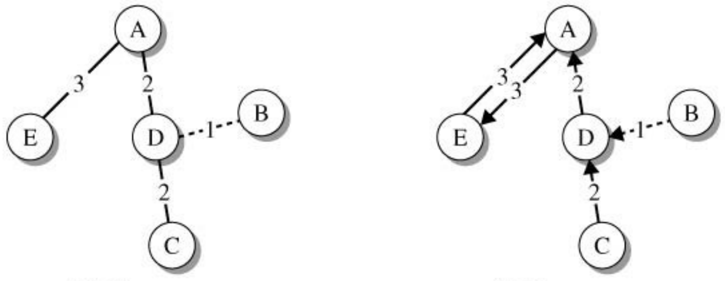
interface GRAPH<T>	ds.util.Graph
	Methods (continued)
int	**numberOfVertices()** Returns the number of vertices in the graph.
boolean	**removeEdge**(T v1, T v2) If (v1,v2) is an edge, removes the edge and returns true; otherwise, returns false. If v1 or v2 is not a vertex in the graph, throws IllegalArgumentException.
boolean	**removeVertex**(Object v) If v is a vertex in the graph, removes it from the graph and returns true; otherwise, returns false.

Creating and Using Graphs (continued)



interface GRAPH<T>	ds.util.Graph
	Methods (continued)
int	**setWeight**(T v1, T v2, int w) If edge (v1, v2) is in the graph, update the weight of the edge and return the previous weight; otherwise, return -1. If v1 or v2 is not a vertex in the graph, throws IllegalArgumentException.
Set<T>	**vertexSet**() Returns a set-view of the vertices in the graph.

Creating and Using Graphs (continued)



addEdge("B", "D", 1)
D is a neighbor of B,
and B is a neighbor of D.

(a)

addEdge("B", "D", 1)
D is a neighbor of B.
B is not a neighbor of D.

(b)

Differences in implementation between a digraph and an undirected graph.

The DiGraph Class

- The DiGraph class implements the Graph interface and adds other methods that are useful in applications.
  - A constructor creates an empty graph.
  - The methods inDegree() and outDegree() are special methods that access a properties that are unique to a digraph.
  - The static method readGraph() builds a graph whose vertices are strings.

The DiGraph Class (continued)

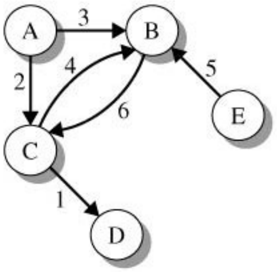
- DiGraph method readGraph() inputs the vertex values and the edges from a textfile.
  - File format:

(Number of Edges n)
Source$_1$ Destination$_1$ Weight$_1$
Source$_2$ Destination$_2$ Weight$_2$
. . .
Source$_n$ Destination$_n$ Weight$_n$

The DiGraph Class (continued)

- The method toString() provides a representation of a graph. For each vertex, the string gives the list of adjacent vertices along with the weight for the corresponding edge. The information for each vertex also includes its in-degree and out-degree.

The DiGraph Class (continued)

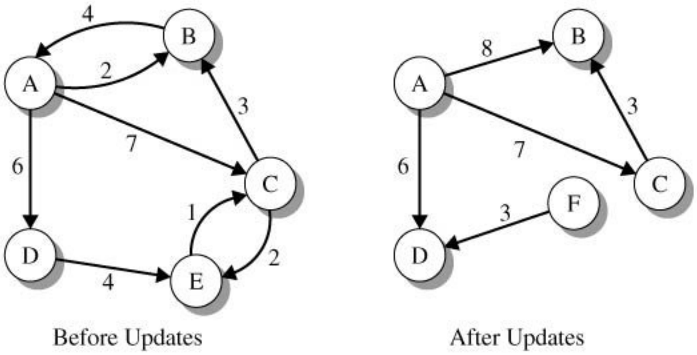


```
File samplegraph.dat
5         // data for the vertices
A B C D E
6         // data for the edges
A B 3
A C 2
B C 6
C B 4
C D 1
E B 5

// input vertices, edges, and weights from samplegraph.dat
DiGraph g = DiGraph.readGraph("samplegraph.dat");

// display the graph
System.out.println(g)
```

The DiGraph Class (continued)

```
Output:
A:  in-degree 0  out-degree 2
    Edges: B(3)  C(2)
B:  in-degree 3  out-degree 1
    Edges: C(6)
C:  in-degree 2  out-degree 2
    Edges: B(4)  D(1)
D:  in-degree 1  out-degree 0
    Edges:
E:  in-degree 0  out-degree 1
    Edges: B(5)
```

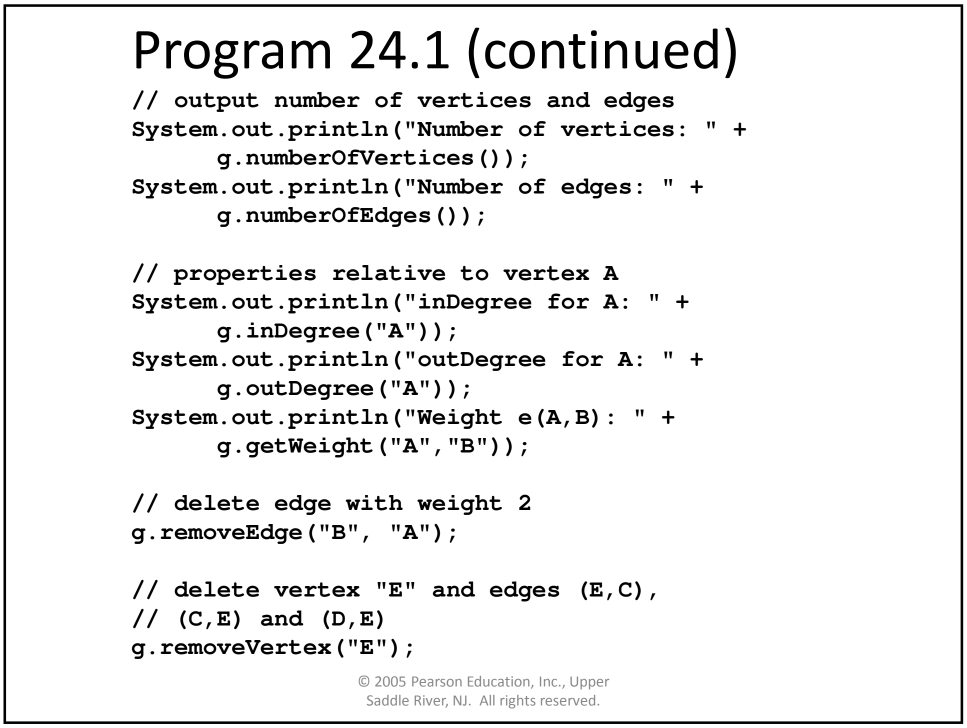
Program 24.1



The figure gives you a view of the graph
before and after the updates occur.

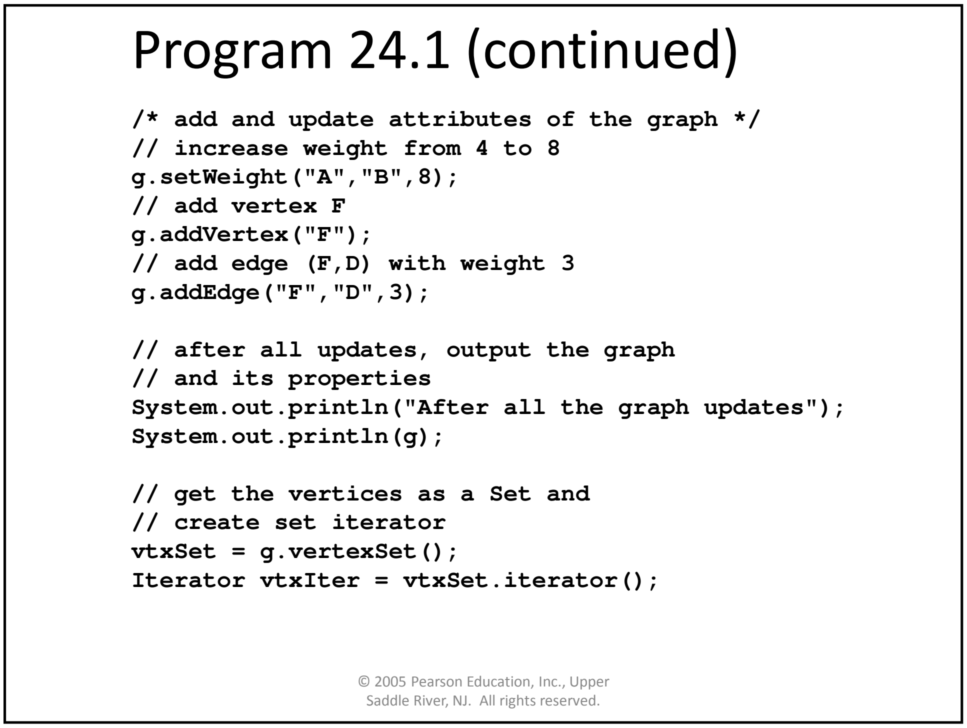
Program 24.1 (continued)

```
import java.io.FileNotFoundException;

import ds.util.Set;
import ds.util.Iterator;
import ds.util.DiGraph;

public class Program24_1
{
   public static void main(String[] args)
   throws FileNotFoundException
   {
      // construct graph with vertices of type
      // String by reading from the file "graphIO.dat"
      DiGraph<String> g =
            DiGraph.readGraph("graphIO.dat");
      String vtxName;
      // sets for vertexSet() and adjacent
      // vertices (neighbors)
      Set<String> vtxSet, neighborSet;
```



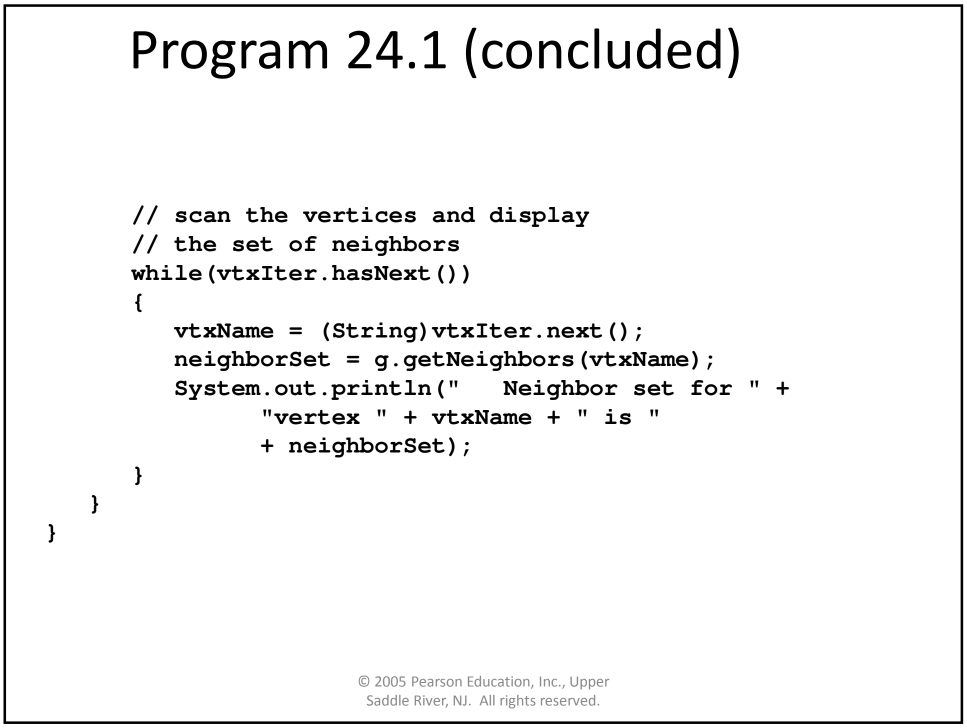
Program 24.1 (continued)

```
// output number of vertices and edges
System.out.println("Number of vertices: " +
      g.numberOfVertices());
System.out.println("Number of edges: " +
      g.numberOfEdges());

// properties relative to vertex A
System.out.println("inDegree for A: " +
      g.inDegree("A"));
System.out.println("outDegree for A: " +
      g.outDegree("A"));
System.out.println("Weight e(A,B): " +
      g.getWeight("A","B"));

// delete edge with weight 2
g.removeEdge("B", "A");

// delete vertex "E" and edges (E,C),
// (C,E) and (D,E)
g.removeVertex("E");
```



Program 24.1 (continued)

```
/* add and update attributes of the graph */
// increase weight from 4 to 8
g.setWeight("A","B",8);
// add vertex F
g.addVertex("F");
// add edge (F,D) with weight 3
g.addEdge("F","D",3);

// after all updates, output the graph
// and its properties
System.out.println("After all the graph updates");
System.out.println(g);

// get the vertices as a Set and
// create set iterator
vtxSet = g.vertexSet();
Iterator vtxIter = vtxSet.iterator();
```



Program 24.1 (concluded)

```
      // scan the vertices and display
      // the set of neighbors
      while(vtxIter.hasNext())
      {
         vtxName = (String)vtxIter.next();
         neighborSet = g.getNeighbors(vtxName);
         System.out.println("   Neighbor set for " +
               "vertex " + vtxName + " is "
               + neighborSet);
      }
   }
}
```